Automatic Music Composition using Answer Set Programming

GEORG BOENN

Cardiff School of Creative & Cultural Industries,
University of Glamorgan,
Pontypridd, CF37 1DL, UK
(*e-mail:* gboenn@glam.ac.uk)

MARTIN BRAIN, MARINA DE VOS and JOHN FFITCH

Department of Computer Science,
University of Bath,
Bath, BA2 7AY, UK
(*e-mail:* {mjb,mdv,jpff}@cs.bath.ac.uk)

submitted 15 June 2009; revised 1 January 2003; accepted 1 January 2003

Abstract

While music composition used to be a pen and paper activity, these days these days music is often composed with the aid of computer software. This can be taken a step further by having the computer compose parts of the score on its own.

The composition of most styles of music is governed by rules. We show that by approaching the automation, analysis and verification of composition as a knowledge representation task and formalising these rules in a suitable logical language, powerful and expressive intelligent composition tools can be easily built.

This paper describes the use of answer set programming to construct an automated system, named Anton, that can compose both melodic and harmonic music, diagnose errors in human compositions and serve as a computer-aided composition tool. The combination of harmonic and melodic composition in a single framework makes Anton unique in the growing area of algorithmic composition.

With real-time composition in some areas, Anton can not only be used as a component in an interactive composition tool but also for live performances and concerts or automatically generated background music in a variety of applications. With the use of a fully declarative language and an "off-the-shelf" reasoning tool, Anton provides a tool for the human composer which is significantly simpler, compact and versatile than other existing systems.

KEYWORDS: Answer set programming, music composition, harmonic and melodic composition, diagnosis

1 Introduction

Music, although it seeks to communicate via emotions, is almost always governed by complex and rigorous rules which provide the base from which artistic expression can be attempted. In the case of musical composition, in most styles there are rules

which describe the progression of a melody, both at the local level (the choice of the next note) and at the global level (the overall structure). Other rules describe the harmony, which arises from the relationship between the melodic line and the supporting instruments.

These rules were developed to guide and support human composers working in the style of their choice, but we wish to demonstrate here that by using knowledge representation techniques, we can create a computer system that can reason about and apply compositional rules. Such a system will provide a simple and flexible way of composing music automatically, but, provided that the representation technology used is sufficiently flexible to allow changes at the level of the rules themselves, it will also help the human composer to understand, explore and extend the rules he is working with.

Composers of music have since the beginning of recorded time used a number of processes to generate the next note, be it simple scales or arpeggios, or complex mathematical structures. Our interest in developing computational systems for composing, harmonising and accompanying music is also old. We have used a variety of mechanisms in searching for a viable system, including encodings of the stochastic and symbolic music of Xenakis (1992), and attempts to find simpler schemes to the major work in artificial intelligence on the harmonisation of Bach chorales by Ebcioğlu (1986).

This paper describes ANTON, an automatic composition system capable of simple melodies with accompaniment, in particular for species one counterpoint as practised in the early Renaissance. The insights gained from this system can be and are being extended to other musical styles, by adding or changing the rules. What has impressed us has been particularly the ease in which musical experience could be converted into code which is succinct and easy to verify.

ANTON uses Answer Set Programming(ASP) (Gelfond and Lifschitz 1988a), a logic programming paradigm, to represent the music knowledge and rules of the system. A detailed description is provided in Section 3, after a short description of the musical aspects of the project (Section 2).

The initial system, ANTON1.0, was first presented in (Boenn et al. 2008). In this paper, we present ANTON1.5. The simplicity of the basic encoding is presented in Section 4. As we will demonstrate, the new version excluding rhythm is significantly faster allowing ANTON to be used in real-time rather than just an interactive tool.

The initial system (Boenn et al. 2008), has only an extremely simple concept of rhythm, all notes having the same length. In Section 5 we show how this restriction has been relaxed, allowing interesting, but "correct" rhythmic patterns. The paper is completed with a discussion of the performance, both musical and computational in Section 6, the use of ASP in Section 7 and future work (Section 8).

Our overall aim is multi-faceted; on one hand we can test the quality and utility of ASP solvers in real-world applications, and on the other develop musicological ideas, create music, and test musical thoughts. We also note that ANTON is usable as part of a student marking system, checking that harmonisations fit the rules. Alternatively, the system can be used as a diagnostic tool or music completion tool, where part of the piece is given to the program to be correctly completed.

2 Music Theory

Music is a world-wide phenomenon across all cultures. The details of what constitutes music may vary from nation to nation, but it is clear that music is an important component of being human.

In the work of this paper we are concentrating on western traditional tonal musics, but the underlying concepts can be translated to other traditions. The particular area of interest here is composition; that is creating new musical pieces.

Creating melodies, that is sequences of pitched sounds, is not as easy as it sounds. We have cultural preferences for certain sequences of notes and preferences dictated by the biology of how we hear. This may be viewed as an artistic (and hence not scientific) issue, but most of us would be quick to challenge the musicality of a composition created purely by random whim. Students are taught rules of thumb to ensure that their works do not run counter to cultural norms and also fit the algorithmically definable rules of pleasing harmony when sounds are played together.

"Western tonal" simply refers to what most people in the West think of as "classical music", the congenial Bach through Brahms music which feels comfortable to the modern western ear because of its adherence to familiar rules. Students of composition in conservatoires are taught to write this sort of music as basic training. They learn to write melodies and to harmonise given melodies in a number of sub-versions. If we concentrate on early music then the scheme often called informally "Palestrina Rules" is an obvious example for the basis of this work. Similarly, harmonising Bach chorales is a common student exercise, and has been the subject of many computational investigations using a variety of methods.

For the start of this work we have opted to work with Renaissance Counterpoint. This style was used by composers like Josquin, Dufay or Palestrina and is very distinct from the Baroque Counterpoint used by composers like Bach, Haendel.

We have used the teaching at one conservatoire in Köln to provide the basic rules, which were then refined in line with the general style taught. The point about generating melodies is that the "tune" must be capable of being accompanied by one or more other lines of notes, to create a harmonious whole. The requirement for the tune to be capable of harmonisation is a constraint that turns a simple sequence (a *monody*) to a *melody*.

Our experience with this work is to realise how many acceptable melodies can be created with only a few rules, and as we add rules, how much better the musical results are.

In this particular style of music complete pieces are not usually created in one go. Composers create a number of sections of melody, harmonising them as needed, and possibly in different ways, and then structuring the piece around these basic sections. Composing between 4 bars and 16 bars is not only a computationally convenient task, it is actually what the human would do, creating components from which the whole is constructed. So although the system described here may be limited in its melodic scope, it has the potential to become a useful tool across a range of sub-styles.

2.1 Automatic Composition

A common problem in musical composition can be summarised in the question "where is the next note coming from?". For many composers over the years the answer has been to use some process to generate notes. It is clear that in many pieces from the Baroque period that simple note sequences are being elaborated in a fashion we would now call algorithmic. For this reason we can say that algorithmic composition is a subject that has been around for a very long time. It is usual to credit Mozart's Musikalisches Würfelspiel (Musical Dice Game) (Chuang 1995) as the oldest classical algorithmic composition, although there is some doubt if the game form is really his. In essence the creator provides a selection of short sections, which are then assembled according to a few rules and the roll of a set of dice to form a Minuet[1]. Two dice are used to choose the 16 minuet measures from a set of 176, and another die selects the 16 trio measures[2], this time from 96 possible. This gives a total number of 1.3×10^{29} possible pieces. This system however, while using some rules, relies on the coherence of the individual measures. It remains a fun activity, and recently web pages have appeared that allow users to create their own original(ish) "Mozart" compositions.

In the music of the second Viennese school ("12-tone", serial music) there is a process in action, rotating, inverting and use of retrograde, but usually performed by hand.

More recent algorithmic composition systems have concentrated on the generation of monody[3], either from a mathematical sequence, chaotic processes, or Markov chains, trained by consideration of acceptable other works. Frequently the systems rely on a human to select which monodies should be admitted, based on judgement rather than rules. Great works have been created this way, in the hands of great talents. Probably the best known of the Markov chain approach is Cope's significant corpus of Mozart pastiche (Cope 2006).

In another variation on this approach, the accompanist, either knowing the chord structure and style in advance, or using machine-listening techniques, infers a style of accompaniment. The former of these approaches can be found in commercial products, and the latter has been used by some jazz performers to great effect, for example by George E. Lewis.

A more recent trend is to cast the problem as one of constraint satisfaction. For example PWConstraints is an extension for IRCAM's Patchwork, a Common-Lisp-based graphical programming system for composition. It uses a custom constraint solver employing backtracking over finite integer domains. OMSituation and OMClouds are similar and are more recently developed for Patchwork's successor OpenMusic. A detailed evaluation of them can be found in (Anders 2007), where the author gives an example of a 1st-species counterpoint (two voices, note against note) after (Fux 1725) developed with Strasheela, a constraint system for music

[1] A dance form in triple time, *i.e.* with 3 beats in each measure
[2] A Trio is a short contrasting section played before the minuet is repeated
[3] A monody is a single solo line, in opposition to homophony and polyphony

built on the multi-paradigm language Oz. Our musical rules however implement the melody and counterpoint rules described by (Thakar 1990), which we find give better musical results.

One can distinguish between *improvisation* systems and *composition* systems. In the former the note selection progresses through time, without detailed knowledge of what is to come. In practice this is informed either by knowing the chord progression or similar musical structures (Brothwell and ffitch 2008), or using some machine listening. In this paper we are concerned with *composition*, so the process takes place out of time, and we can make decisions in any order.

It should also be noted that these algorithmic systems compose pieces of music of this style in either a melodic or a harmonic fashion, and are frequently associated with computer-based synthesis. The system we will propose later is unique as it deals with both simultaneously.

2.1.1 Melodic Composition

In melodic generation a common approach is the use of some kind of probabilistic finite state automaton or an equivalent scheme, which is either designed by hand (some based on chaotic oscillators or some other stream of numbers) or built via some kind of learning process. Various Markov models are commonly used, but there have been applications of n-grams, genetic algorithms and neural nets. What these methods have in common is that there is no guarantee that melodic fragments generated have acceptable harmonic derivations. Our approach, described below is fundamentally different in this respect, as our rules cover both aspects simultaneously.

In contrast to earlier methods, which rely on learning, and which are capable of giving only local temporal structure, a common criticism of algorithmic melody (Leach 1999), we do not rely on learning and hence we can aspire to a more global, whole melody, approach. In addition we are no longer subject to the limitations of the kind of process which, because it only works in time in one direction, is hard to use in a partially automated fashion; for example operations like "fill in the 4 notes between these sections" is not a problem for us.

We are also trying to move beyond experiments with random note generation, which we have all tried and abandoned because the results are too lacking in structure. Predictably, the alternative of removing the non-determinism at the design stage (or replacing with a probabilistic choice) runs the risk of 'sounding predictable'! There have been examples of good or acceptable melodies created like this, but the restriction inherent in the process means it probably works best in the hands of geniuses.

2.1.2 Harmonic Composition

A common usage of algorithmic composition is to add harmonic lines to a melody; that is notes played at the same time as the melody that are in general consonant and pleasing. This is exemplified in the harmonisation of 4-part chorales, and has

been the subject of a number of essays in rule-based or Markov-chain systems. Perhaps a pinnacle of this work is (Ebcioğlu 1986) who used early expert system technology to harmonise in the style of Bach, and was very successful. Subsequently there have been many other systems, with a range of technologies. There is a review included in (Rohrmeier 2006).

Clearly harmonisation is a good match to constraint programming based systems, there being accepted rules[4]. It also has a history from musical education.

But these systems all start with a melody for which at least one valid harmonisation exists, and the program attempts to find one, which is clearly soluble. This differs significantly from our system, as we generate the melody and harmonisation together, the requirement for harmonisation affecting the melody.

3 Answer Set Programming

In *answer set programming* (Baral 2003) a logic program is used to describe the requirements that must be fulfilled by the solutions of a certain problem. The answer sets of the program, usually defined through (a variant/extension of) the stable model semantics (Gelfond and Lifschitz 1988b), then correspond to the solutions of the problem. This technique has been successfully applied in domains such as planning (Eiter et al. 2002; Lifschitz 2002), configuration and verification (Soininen and Niemelä 1999), super-optimisation (Brain et al. 2006), diagnosis (Eiter et al. 1999), game theory (De Vos and Vermeir 1999) multi-agent systems (Baral and Gelfond 2000; Buccafurri and Gottlob 2002; De Vos and Vermeir 2004; Buccafurri and Caminiti 2005; Cliffe et al. 2006), reasoning about biological networks (Grell et al. 2006), voting theory (Konczak 2006), policy mechanisms(Mileo and Schaub 2006), generation of phylogenetic trees (Erdem et al. 2006), evolution of language (Erdem et al. 2003) and game character descriptions (Padovani and Provetti 2004).

There is a large body of literature on ASP: for in-depth coverage see (Baral 2003; Gelfond and Lifschitz 1988b), but for the sake of making this paper self-contained we will cover the essentials as they pertain to our usage here.

Basic Concepts: The answer set semantics is a model based semantics for normal logic programs. Following the notation of (Baral 2003), we refer to the language over which answer set semantics is defined as *AnsProlog*.

The smallest building block of an *AnsProlog*is an atom or predicate, e.g. `r(X,Y)` denotes XrY. X and Y are variables which can be grounded with constants. Each ground atom can be assigned the truth value *true* or *false*.

For current purposes, we only require answer sets with one type of negation, namely negation-as-failure denoted **not**. This type of negation states that something should be assumed false when it cannot be proven to be true. A literal is an atom a or its negation **not** a. We extend the notation to sets: **not** S is the set $\{\textbf{not}\ l \mid l \in S\}$ with S a set of literals.

An *AnsProlog* program consist of a finite set of statements, called rules. Each

[4] For example see: `http://www.wikihow.com/Harmonise-a-Chorale-in-the-Style-of-Bach`

rule $r : a \leftarrow B.$ or $\bot \leftarrow B.$ is made of two parts namely the body B, denoted $B(r)$, which is a set of literals, and a head atom a or $\bot$, denoted $H(r)$. The body can be divided in two parts: the set of positive atoms, denoted as $B^+(r)$, and the set of negated atoms, denoted $B^-(r)$. A rule should be read as: "a is *supported* if all elements of B are true". A rule with empty body is called a *fact* and we often only mention the head. A rule with head $\bot$ is referred to as an *(integrity) constraint*. We often omit the $\bot$ symbol and leave the head empty. $\bot$ is always assigned the truth value "false". A program is called positive if it does not contain any negated atoms.

The finite set of all constants that appear in the program P is referred to as the *Herbrand universe*, denoted $\mathcal{U}_P$. Using the Herbrand universe, we can ground the entire program. Each rule is replaced by its grounded instances, which can be obtained by replacing each variable symbol by an element of $\mathcal{U}_P$. The ground program, denoted $ground(P)$, is the union of all ground instances of the rules in P.

The set of all atoms grounded over the Herbrand universe of a program is called the *Herbrand base*, denoted as $\mathcal{B}_P$. These are exactly those atoms that will appear the grounded program.

An assignment of truth values to all atoms in the program (or all elements from the Herbrand base), without causing contradiction, is called an interpretation. Often only those literals that are considered true are mentioned, as all the others are false by definition (negation as failure).

Given a ground rule r, we say a r is *applicable* w.r.t. an interpretation $I \subseteq \mathcal{B}_P$ if all the body elements are true ($B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$). The rule is *applied* w.r.t. I when it is applicable and $H_r \in I$. A ground rule is *satisfied* w.r.t. an interpretation I if it is either not applicable or applied w.r.t. I. An atom is *supported* w.r.t. I if there is an applied rule with this atom in the head. Obviously, we want to make sure that interpretations satisfy every rule in the program. So, an interpretation I is a *model* for a program P iff all rules in $ground(P)$ are satisfied.

To find actual solutions, models alone are not sufficient. To prevent this, we need to make sure that only those literals that are supported are considered true. This results in the so-called minimal model semantics. A model M for a program P is *minimal* if no other model N exists such that $N \subset M$. Programs can have any number of minimal models, while programs without constraints will always admit at least one. Positive programs will have at most one, and exactly one when they do not admit any constraints.

The minimal model of a positive program without constraints can be found using a fixpoint, called the *deductive closure*, which can be computed in polynomial time. We start with the empty set and find all atoms that are supported. With this new set we continue to find supported atoms. When the set no longer changes, we have found the deductive closure or minimal model of the program. If the specification contains constraints, we can follow the same principle but the process fails when a unsatisfied constraint is found.

Definition 1

Let P be a positive *AnsProlog* program and let I be an interpretation. We define

the *immediate consequence operator* $\mathrm{T_P}$ as:

$$\mathrm{T_P}(I) = \{a\mathcal{B}_P \mid \exists\, r \in P \cdot B_r \subseteq I\}$$

$\mathrm{T_P}$ is monotonic so it has a least fixpoint: the deductive closure.

Definition 2
Let P be a positive *AnsProlog* program. The *deductive closure* of P, is the least fixpoint ${\mathrm{T_P}}^{\omega}(\emptyset)$ of $\mathrm{T_P}$.

While the minimal model semantics gives the required solutions with a positive program, it is, in the presence of negation-as-failure, insufficient. A simple example of such a program is: $\{\mathtt{a} : -\mathbf{not}\ \mathtt{a}.; \mathtt{b} : -\mathtt{a}\}$. This program has one minimal model $\{a, b\}$, while the truth of a depends on a being false. Negation-as-failure gives us no guarantee that something is indeed false and that information derived from it is actually correct. To obtain intuitive solutions, we need to verify that our assumptions are indeed correct. This is done by reducing the program to a simpler program containing no instances of negation-as-failure. Given an interpretation, all rules that contain **not** l that are considered false are removed while the remaining rules only retain their literals. This reduction is often referred to as the Gelfond-Lifschitz transformation (Gelfond and Lifschitz 1988a; Gelfond and Lifschitz 1991). When this program gives the same supported literals as the ones with which we began, we have found an answer set.

Definition 3
Let P be a ground *AnsProlog* program. The *Gelfond-Lifschitz transformation* of P w.r.t S, a set of ground atoms is the program P^S containing the rules $l \leftarrow B$ such that $l \leftarrow B, \mathbf{not}\ C \in P$ with $C \cap S = \emptyset$, and B and C are sets of literals.

Definition 4
Let P be a *AnsProlog* . A set of ground atoms $S \subseteq \mathcal{B}_P$ is an *answer set* of P iff S is the minimal model of $ground(P^S)$.

The uncertain nature of negation-as-failure gives rise to several answer sets, which are all acceptable solutions to the problem that has been modelled. It is in this non-determinism that the strength of answer set programming lies.

Extensions The basic formalism, single head and negation-as-failure only appearing in the head, already enables the representation of many problems. However, for some applications the programmer is forced to write code in a more round-about non intuitive way. To overcome this, extensions are introduced.

From a programmer's perspective, choice rules(Niemelä et al. 1999) and symbolic functions are probably the most commonly used extensions. A lot of problems require choices between a set of atoms to made. Although this can be modelled in the basic formalism it tends to increase to the number of rules and increases the possibility of errors. To solve this, choices are introduced. Choices written $L\{l_1, \ldots l_n\}M$ are a convenient construct to indicate that at least L and at most M from the set $\{l_1\ l_n\}$ must be true in order to satisfy the construct. L defaults to 0 while M

defaults to n. Choice rules are often used in conjunction with a grounding predicate: $L\{A(X) : B(X)\}M$ represents the choice of a number of atoms $A(X)$ where is grounded with all values of X for which $B(X)$ is true.

A symbolic function `f(X,Y)` defines a new constant that is the value of the function. It is used as a shorthand to group sets of variables together in a meaningful way. An example could be to group coordinates together: `position(X,Y)`. This can then be used like: `rectangle(position(2,5),position(5,8))`.

Implementations: Algorithms and implementations for obtaining answer sets of logic programs are referred to as *answer set solvers*. The most popular and widely used solvers are DLV(Eiter et al. 1998) and SMODELS(Niemelä and Simons 1997) and more recently CLASP (Gebser et al. 2007a).

Alternatives are CMODELS(Giunchiglia et al. 2004) and SUP(Lierler 2008), solvers based on translating the program to a SAT problem, and SMODELS-IE(Brain et al. 2007), the cache-efficient version of SMODELS. Furthermore, there is the distributed solver PLATYPUS (Gressmann et al. 2005).

To solve a problem it first needs to be grounded. Currently three grounders are used: the grounder supplied with DLV, LPARSE, the grounder that was developed together with SMODELS but is used by most solvers, and the very recent one GRINGO (Gebser et al. 2007) which works together with CLAPS. During the grounding phase, not only are the variables substituted for constants, but also the rules that lead to nothing are eliminated. Furthermore, the grounders try to optimise the program as much as possible. The second phase is solving which takes a grounded program as input and generates the set of its answer sets.

Current answer set solvers can be divided in three groups depending on the style of algorithm they use or the mapping the use: branch-and-bound (Eiter et al. 1998; Niemelä and Simons 1997; Brain et al. 2007; Gressmann et al. 2005), clause learning (Giunchiglia et al. 2004; Gebser et al. 2007a) and a mapping to SAT (Giunchiglia et al. 2004; Lierler 2008). All use a variety of heuristics to improve the performance of the basic algorithm. In this paper, we will only provide a very high-level overview of both ASP-only algorithms.

Branch and bound is a bottom-up approach working with partial interpretation containing those atoms that are considered true and those that are false. Each iteration consists of two phases: branch and bound. The latter considers, using the partial information, which atoms have to true and which ones should be false based on the support or lack thereof already generated by this partial interpretation. When no further information can be deduced, the branch phase selects an atom which truth value is the branching point. One branch continue with this atom true and the other which the atom false. The process terminates if the truth value of all atoms in the program is determined.

In clause learning, the *AnsProlog* program is translated in a collection of logic clauses using Clark's completion. The solver will then try to find a truth assignments that satisfies all clauses in a bottom up fashion. Each time an inconsistency is found. This new constraint may reduce the search space, as future partial evaluations may be found inconsistent without further search.

4 Anton

4.1 System Description

Anton applies ASP techniques to compositional rules to produce an algorithmic composition system. *AnsProlog* is used to write a description of the rules that govern the melodic and harmonic properties of a correct piece of music; in this way the program works as a model for music composition that can be used to assist the composer by suggesting, completing and verifying short pieces.

The composition rules are modelled so that the *AnsProlog* program defines the requirements for a piece to be musically valid, and thus every answer set corresponds to a different valid piece. To generate a new piece the composition system simply has to generate an (arbitrary) answer set. Rather than the traditional problem/solution mapping of answer set programming, this is using an *AnsProlog* program to create a 'random' (arbitrary) example of a complex, structured object.

In this section we will only discuss the basic system of Anton1.5.

The *AnsProlog* program of the basic system is divided among various files: notes.lp, modes.lp, chord.lp, progression.lp, melody.lp and harmony.lp. The first contains the general background rules on notes and intervals while the second describes the various modes/keys the system can use and their consequences for note selection and position. The current system is able to work with major, minor, Dorian, Lydian and Phrygian keys. Chord.lp provides the description of chords and chordal progression and the effects of node choices. Rules for the progression of all parts, either melodic and harmonic, are handled in progression.lp These part of the program is responsible for selecting the next node in each of the parts on the basis of a previous note. The rules for melodic parts and for working with multiple parts are encoded in melodic.lp and harmonic.lp respectively.

We will discuss progression, melody and harmony in more detail. The whole system is licensed under the GPL and publicly available via `http://www.cs.bath.ac.uk/~mjb/anton/`.

Figure 1 presents a selection of rules dealing with progression of nodes. The model is defined over a number of time steps, given by the variable `T`. The key proposition is `chosenNote(P,T,N)` which represents the concept "At time `T`, part `P` plays note `N`". To encode the options for melodic progress ("the tune either steps up or down one note in the key, leaps more than one note, repeats or rests"), choice rules are used. For diagnostic and debugging purposes, we decided not to encode compositional errors immediately as constraints, but instead use error rules like `error(P,T,err_ip) :- incorrectProgression(P,T)`. Using a constraints to include answer sets with error-atoms or excluding them entirely, we can alter the functionality of our system without changing the code. We will later return to various uses of our system.

To encode the melodic limits on the pattern of notes and the harmonic limits on which combinations of notes may be played at once, error-rules like the one in progression.lp are included. Figure 2 shows how we encoded rules that forbid repetition of notes in the melodic parts, octave leaps except for special circumstances, impulses, repetition of more than two notes and certain lengths of intervals. While

```
%% Each part picks at most one note per time step
time(1..t).

%% Each part can only play one note at a given time
  :- 2  choosenNote(P,T,NN) : note(NN), rest(P,T) .

%% At every time step the note may change
%% It changes by stepping (moving one note in the scale)
%% or leaping (moving more than one note)
%% These can either be upwards or downwards
1 { changes(P,T), repeated(P,T), toRest(P,T), fromRest(P,T),
  incorrectProgression(P,T) } 1 :- T != t.
1 { stepAt(P,T), leapAt(P,T) } 1 :- changes(P,T), T != t.
1 { downAt(P,T), upAt(P,T) } 1 :- changes(P,T), T != t.

stepDown(P,T) :- stepAt(P,T), downAt(P,T).
stepUp(P,T) :- stepAt(P,T), upAt(P,T).

#const err_ip="Incorrect progression".
reason(err_ip).
error(P,T,err_ip) :- incorrectProgression(P,T).

%% If we step, we must pick an amount to step by
1 { stepBy(P,T,SS) : stepSize(SS) : SS < 0 } 1 :- stepDown(P,T).
1 { stepBy(P,T,SS) : stepSize(SS) : SS > 0 } 1 :- stepUp(P,T).

%% Make it so
choosenNote(P,T + 1,N + S) :- choosenNote(P,T,N), stepAt(P,T), stepBy(P,T,S), note(N + S).
choosenNote(P,T + 1,N + L) :- choosenNote(P,T,N), leapAt(P,T), leapBy(P,T,L), note(N + L).
choosenNote(P,T + 1,N) :- choosenNote(P,T,N), repeated(P,T).
```

Fig. 1. A code fragment from progression.lp

some of these rules might be valid in other types of music, renaissance counterpoint explicitly forbids them.

Interaction between parts is governed by the harmony. Figure 3 show how we encoded the musical rules that specify that you cannot have dissonant intervals between parts, limit the distance between parts and that parts cannot cross-overs.

While the fragments shown in Figures 1-3 are only a selection of the entire, they demonstrate that the rules are very simple and intuitive (with the necessary musical background). The modelling of this style of music, excluding rhythm, contains less than 200 ungrounded logic rules.

4.2 Features

In the previous section we discussed the basic components of the ANTON system. However, in order to have a complete system, we are still missing one component: the specification of parts. Currently, the system comes with descriptions for solos, duets, trios and quartets but the basic system is written with no fixed number of parts in mind. Figure 4 shows the description for a quartet.

Depending on how the system is used, composition or diagnosis, you will either be interested in those pieces that do not result in errors at all, or in an answer set that mentions the error messages. For the former we simply specify the constraint `:- error(P,T,R).` effectively making any error rule into a constraint. For the latter we include the rules: `errorFound :- error(P,T,R).` and `:- not errorFound.` requiring that an error is found (i.e. returning no answers if the diagnosed piece is error free).

```
%% Melodic parts are not allowed to repeat notes
#const err_nrmp="No repeated notes in melodic parts".
reason(err_nrmp).
error(MP,T,err_nrmp) :- repeated(MP,T).

#const err_olnf="Leap of an octave from a note other than the fundamental".
reason(err_olnf).
error(MP,T,err_olnf) :- leapBy(MP,T,12), not choosenChromatic(MP,T,1).
error(MP,T,err_olnf) :- leapBy(MP,T,-12), not choosenChromatic(MP,T,1).

%% Impulse %% Stepwise linear progression creates impulse
%% Leaps create impulse - using the notes in between resolves this
downwardImpulse(MP,T+1) :- leapDown(MP,T), time(T+1).
downwardImpulse(MP,T+3) :- stepDown(MP,T+2), stepDown(MP,T+1), stepDown(MP,T), time(T+3).
upwardImpulse(MP,T+1) :- leapUp(MP,T), time(T+1).
upwardImpulse(MP,T+3) :- stepUp(MP,T+2), stepUp(MP,T+1), stepUp(MP,T), time(T+3).

%% No repetition of two or more notes
#const err_rn="Repeated notes".
reason(err_rn).
error(MP,T1,err_rn) :- choosenNote(MP,T1,N), stepBy(MP,T1,S1),
             choosenNote(MP,T2,N), stepBy(MP,T2,S1),
             T1 + 1 < T2, T2 < T1 + 2 + RW.
error(MP,T1,err_rn) :- choosenNote(MP,T1,N), leapBy(MP,T1,L1),
choosenNote(MP,T2,N), leapBy(MP,T2,L1),
T1 + 1 < T2, T2 < T1 + 2 + RW.
#const err_dc="Dissonant contour".
reason(err_dc).
error(MP,t,err_dc) :- lowestNote(MP,N1), highestNote(MP,N2),
             chromatic(N1,C1), chromatic(N2,C2),
             not consonant(C1,C2), N1 < N2.
```

Fig. 2. A code fragment from melody.lp

```
#const err_dibp="Dissonant interval between parts".
reason(err_dibp).
error(P1,T,err_dibp) :- choosenChromatic(P1,T,C1), choosenChromatic(P2,T,C2),
             P1 < P2, chromaticInterval(C1,C2,D),
             not validInterval(D).

%% The maximum distance between parts is an octave plus 4 semitones (i.e. 16 semitones).
#const err_mdbp="Over maximum distance between parts".
reason(err_mdbp).
error(P,T,err_mdbp) :- choosenNote(P,T,N1), choosenNote(P+1,T,N2),
             N1 > N2 + 16, part(P+1).

%% Parts cannot cross over.
#const err_pcc="Parts can not cross".
reason(err_pcc).
error(P,T,err_pcc) :- choosenNote(P,T,N1), choosenNote(P+1,T,N2),
             N1 < N2, part(P+1).
```

Fig. 3. A code fragment from harmony.lp

These simple rules are encoding in composing.lp and diagnosis.lp so that can be included when our scripts assemble to program.

By adding constraints on which notes can be included, it is possible to specify part or all of a melody, harmony or complete piece. This allows ANTON to be used for a number of other tasks beyond automatic composition. By fixing the melody it is possible to use it as an automatic harmonisation tool. By fixing part of a piece, it can be used as computer aided composition tool. By fixing a complete piece, it is possible to check its conformity with the rules, for marking student compositions or harmonisations. Alternatively we could request the system to complete part of a piece. In order to do so, we provide the system with a set of *AnsProlog* facts

```
%% This is a quartet
style(quartet).

%% There are four parts
part(1..4).

%% The top part plays the melody
melodicPart(1).

%% For chords we need to know the lowest part
lowestPart(4).

%% We need a range of up to 2 octaves (24 steps) for each part,
%% thus need 24 notes above and below the lowest / highest start
#const quartetBottomNote=1.
#const quartetTopNote=68.
note(quartetBottomNote..quartetTopNote).
bottomNote(quartetBottomNote).
topNote(quartetTopNote).

%% Starting positions are 1 - 5 - 1 - 5
#const err_isn="Incorrect starting note".
reason(err_isn). error(1,1,err_isn) :- not choosenNote(1,1,44).
error(2,1,err_isn) :- not choosenNote(2,1,37).
error(3,1,err_isn) :- not choosenNote(3,1,32).
error(4,1,err_isn) :- not choosenNote(4,1,25).

%% No rests
#const err_nrfw="No rest for the wicked".
reason(err_nrfw).
error(P,T,err_nrfw) :- rest(P,T).

%% With three or more parts allow intervals of a major fourth
%% (5 semitones) between parts
validInterval(5).
```

Fig. 4. The quartet specification

expressing the mode (major, minor, etc.), the notes which are already fixed, the number of notes in the piece, the configuration and the number of parts.

The complete system consists of three major phases; building the program, running the ASP program and interpreting the results. As a simple example suppose we wish to create a 4 bar piece in E major one would write

```
programBuilder.pl --task=compose --mode=major --time=16 > program
```

which builds the ASP program, giving the length and mode. Then

```
gringo < program | ./shuffle.pl 6298 | claps 1 > tunes
```

runs the ASP phase and generates a representation of the piece. We provide a number of output formats, one of which is a CSOUND (Boulanger 2000) program with a suitable selection of sounds.

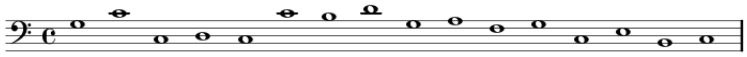
```
$ parse.pl --fundamental=e --output=csound < tunes > tunes.csd
```

generates the Csound input from the generic format, and then

```
$ csound tunes.csd -o dac
```

plays the melody. We provide in addition to Csound, output in human readable format, ASP facts or the Lilypond score language. Figure 5 shows the score of the tunes piece composed above.

Alternatively we could request the system to complete part of a piece. In order to do so, we provide the system with a set of ASP facts expressing the keyMode, the



```
| 55 60 48 50 48 60 59 62 55 57 53 55 48 52 47 48
|  G  C  C  D  C  C  B  D  G  A  F  G  C  E  B  C
|   +5 -12 +2 -2 +12 -1 +3 -7 +2 -4 +2 -7 +4 -5 +1
```

Fig. 5. The score and human readable format for the tunes composition.

```
keyMode(lydian).
choosenNote(1,1,25).
choosenNote(1,2,24).
choosenNote(1,8,19).
choosenNote(1,9,20).
choosenNote(1,10,24).
choosenNote(1,14,29).
choosenNote(1,15,27).
choosenNote(1,16,25).
#const t=16.
style(solo).
part(1).
```

Fig. 6. musing.lp: An example of a partial piece

notes which are already fixed, the number of notes in your piece, the configuration and the number of parts. Figure 6 contains an example of such file. The format is the same as the one returned from the system except that all the notes in the piece will have been assigned.

We then run the system just as before with the exception of adding `--piece=musing.lp` when we run programBuilder.pl. The system will then return all possible valid composition that satisfy the criteria set out in the partial piece.

The *AnsProlog* programs used in ANTON contains just 191 lines (not including comments and empty lines) and encodes 28 melodic and harmonic rules. Once instantiated, the generated programs range from 3,500 atoms and 13,400 rules (a solo piece with 8 notes) to 11,000 atoms and 1,350,000 rules (a 16 note duet). It should be noted that the 500 lines of code here contrast with the 8000 lines in Strasheela(Anders 2007) and 88000 in Bol(Bel 1998).

5 Rhythm

5.1 Musical Discussion

Our system used to be limited in terms of one of the most essential musical parameters: Rhythm. All music ANTON generated so far was based on rules for classical polyphony, i.e. the combination of musically unique, independent melodic voices, but all events had the same time interval. Within this restriction we are able to generate first-species counterpoint up to four independent voices and solo melodies as well as homophonic 4-part chorales where the rules for the inner voices could be more relaxed from the strict melodic rules that determine each of the four voices in

a polyphonic setting. To expand our system, we are currently implementing temporal interval logic using the classification of Allen (1983), of which an overview is also given in Goranko et al. (2003). The musical areas of harmony and classical counterpoint often have rules about the conditions when a particular constellation of notes may occur on the timeline, sometimes with inclusion of the events preceding and following. For example, a *suspended fourth* is a dissonant constellation of two or more voices created on a strong beat. The voice that suspends the consonant note, i.e. the fourth note of the scale suspending the third, needs to be sounding before the dissonance occurs. This sound needs also to be consonant. After the dissonance on the strong beat the suspending voice needs then to resolve on a weak time interval within the measure. That sound again must be a consonance. To help expressing such rules the interval logic of 'start', 'end', 'during' and 'overlap' needs to be implemented. In music, however, the time intervals themselves are never 'neutral'. There are typical alternations of strong and weak beats in nearly every musical style that is based on an underlying pulse. Those alternating beat patterns form the notion of meter. But, the concept of meter in Renaissance style is a particular one and not the same as the concept of bars, beats and time signatures that is prevailing since the Baroque area and which is still in use today.

There are generally four different kinds of musical time in the Renaissance. When looking at the subdivision of the brevis, which translates into today's double whole-note, there are four different options for the composer of that area, as documented in the famous treatise 'Ars Nova' by Philippe de Vitry. The brevis can be interpreted either as 3 or 2 semi-breve (today's whole note) and those further into 3 and 2 subdivisions called minims (today's half note), so we can subdivide the longa into 3 x 2, 2 x 2, 3 x 3 or 2 x 3 minims[5]. Those different subdivisions were indicated in the vocal score using different time-signatures[6]. The 15th century composer Johannes Ockeghem wrote his *Missa prolationem*, a four-voiced polyphonic masterpiece, using all four different time-signatures together, a different one for each voice, while at the same time the voices imitate their lines containing the same melodic material using canons and double-canons[7] This complex polyrhythmic structure establishes a proportion of note durations as 6:4:9:6 between the voices. The effect of superimposing different beat qualities can be seen in Table 1. Only the smaller note durations below the minim are not affected by the proportional scaling. They remain at the same length for all four voices regardless of their time-signature. When transcribed nowadays into common practice notation the voices would need 36 half notes to complete one large cycle of accent patterns before starting over with a common 'downbeat' (de la Motte 1981). The complete cycle therefore translates

[5] The order of subdivision is musically important in terms of accentuation and therefore has consequences for the treatment of consonance and dissonance. Hence, 3 x 2 is not the same as 2 x 3.

[6] The four time signatures are called *tempus perfectum cum prolationem imperfecta* (3 x 2), *tempus imperfectum cum prolationem imperfecta* (2 x 2), *tempus perfectum cum prolationem perfecta* (3 x 3) and *tempus imperfectum cum prolationem perfecta* (2 x 3)

[7] A canon is a special case of a line that can be imitated simultaneously by another voice n beats later while their combination at the same time satisfies the rules on consonance/dissonance. Temporal interval logic will allow us to implement canons in future versions of Anton.

	'X' marks a downbeat, 'O' marks a 2nd level beat, 'o' marks a 3rd level beat																		
S	X	o	O	o	O	o	X	o	O	o	O	o	X	o	O	o	O	o	
A	X	o	O	o	X	o	O	o	X	o	O	o	X	o	O	o	X	o	
T	X	o	o	O	o	o	O	o	o	X	o	o	O	o	o	O	o	o	
B	X	o	o	O	o	o	X	o	o	O	o	o	X	o	o	O	o	o	
S	X	o	O	o	O	o	X	o	O	o	O	o	X	o	O	o	O	o	X
A	O	o	X	o	O	o	X	o	O	o	X	o	O	o	X	o	O	o	X
T	X	o	o	O	o	o	O	o	o	X	o	o	O	o	o	O	o	o	X
B	X	o	o	O	o	o	X	o	o	O	o	o	X	o	o	O	o	o	X

Table 1. *Polyrhythmic structure of downbeats from Ockeghem's Missa prolationem creating a hyper-meter of 36 beats*

into the filtered Farey Sequence F'_{36}, where each of the ratios in the range of [0...1] denotes the onset time of a half note, see the representation for one voice, the Bass, in Table 2. According to the time-signature (3 x 2) of that particular voice there are three distinct metrical layers that govern the note events occurring on such a grid: The top layer gives us the occurrence of the downbeat, which is the event of most metrical importance, i.e. no dissonance may occur here unless the voice causing it prepares the note on the preceding beat. The second layer denotes the beat level. Although of less metrical weight, no dissonant interval should sound here unless it is a prepared suspension. The third layer provides the lightest metrical events. Here dissonances may occur, for instance in form of a passing note. The rule demanding preparation of the dissonance is relaxed on this level and on all further subdivisions underneath.

The hierarchical pattern of one of Ockeghem's voices in Table 2 shows also some interesting properties that lead to a general method for the construction of such tables. Starting with the first metrical level (I) in Table 2, we see that the largest denominator is always equal to the number of measures required to come back to square one with all the other voices, i.e the number of measures per hyper-meter. All the other denominators on level I are divisors of the largest denominator. The next level (II) shows us how many subdivisions are contained in the hyper-meter on that particular level. This is again indicated by the largest denominator of the level and all other denominators are his divisors *excluding* those already contained on all lower indexed levels. This principle repeats itself on all higher indexed metrical levels. For each denominator n in the scheme, the numerators also follow a simple principle: they traverse the complete ordered list of numbers co-prime to n.

As we have shown in Boenn (2007) and in Boenn (2008), the Farey Sequence is ideal for tasks like rhythmic modelling, music performance analysis and music theory. Hardy and Wright (1938) on pp.23 gave a description and proved the properties of the Farey Sequence. Its scalability and general independence from the concepts of bars and meter is of advantage because it can be applied to numerous different musical styles. We have given a glimpse in the above Renaissance example (Table 2) how it could be used to encode polyrhythmic structures. Other examples can include African Polyrhythm, Western Classical, Avantgarde and Popular Music,

	Tempus perfectum cum prolatione imperfecta - 6 measures x 3 x 2 half notes																							
I	$\frac{0}{1}$									$\frac{1}{6}$										$\frac{1}{3}$					
II			$\frac{1}{18}$			$\frac{1}{9}$						$\frac{2}{9}$			$\frac{5}{18}$							$\frac{7}{18}$			$\frac{4}{9}$	
III		$\frac{1}{36}$			$\frac{1}{12}$			$\frac{5}{36}$			$\frac{7}{36}$			$\frac{1}{4}$			$\frac{11}{36}$			$\frac{13}{36}$			$\frac{5}{12}$			$\frac{17}{36}$
I	$\frac{1}{2}$									$\frac{2}{3}$										$\frac{5}{6}$					
II			$\frac{5}{9}$			$\frac{11}{18}$						$\frac{13}{18}$			$\frac{7}{9}$							$\frac{8}{9}$			$\frac{17}{18}$	
III		$\frac{19}{36}$			$\frac{7}{12}$			$\frac{23}{36}$			$\frac{25}{36}$			$\frac{3}{4}$			$\frac{29}{36}$			$\frac{31}{36}$			$\frac{11}{12}$			$\frac{35}{36}$

Table 2. *Metrical hierarchy from Ockeghem's Missa prolationem expressed as* F'_{36}

Greek Verse Rhythms, Indian Percussion and many more. The principle remains always the same.

Farey Sequences are ordered lists of integer ratios in their lowest terms, of increasing complexity the longer the sequence becomes, and with all ratios in the range of $[0, 1]$. As an example, Figure 7 shows a plot of F_{17} that correlates the position of the ratios in the interval $(0, 1)$ with the unit fraction built from their corresponding denominators. The visualisation clearly points out that the smaller the denominator the larger are the symmetrical gaps around the x-position of the ratio, i.e. the smaller b of the ratio a/b, the greater the distance. We believe that it is due to those relatively large gaps surrounding simpler ratios that they form perceptually useful zones of attraction for more complex ratios that fall into them or that come close enough. It is left for future field studies to measure and to find evidence whether these zones of attraction are perceptually relevant or not. Composers who want to "stay away" from those simple ratios will need to leave a considerable amount of space around these zones of attraction. It becomes also clear that there are various accelerandi and ritardandi encoded in every F_n, for example there are Gestalten that form visible triangles between larger reciprocals and smaller ones in their surrounding area (Figure 7). These Gestalten are formed by monotonically increasing or decreasing values of the denominators. The increasing or decreasing tendencies overlap each other. The gaps between the ratios forming those triangles are always on a logarithmic scale, hence the impression of accelerated or reduced tempo that becomes evident through the sonification of these triangular Getstalten. It is well known that timed durations need to be placed on a logarithmic rather than linear scale in order to convey a "natural" sense of spacing and tempo modification. These structures are clearly mirrored around the $1/2$ value that is part of every F_n with $n > 1$.

The great variety of styles and concepts from medieval to modern eras can be realised by applying intelligent filtering methods to sieve through the sequences. Examples for filtering include probabilistic methods and filters exploiting the prime number composition of integers and ratios, for example b-smooth numbers, or Clarence Barlow's function for the 'Indigestibility' of a natural integer (Hajdu 1993). The Farey Sequence has been known for a while in the area of musical tuning systems[8]. Its use for rhythmic modelling has not been fully exploited yet; ANTON is

[8] for example Erv Wilson's annotations of tunings used by Partch (1979) http://www.anaphoria.com/wilson.html

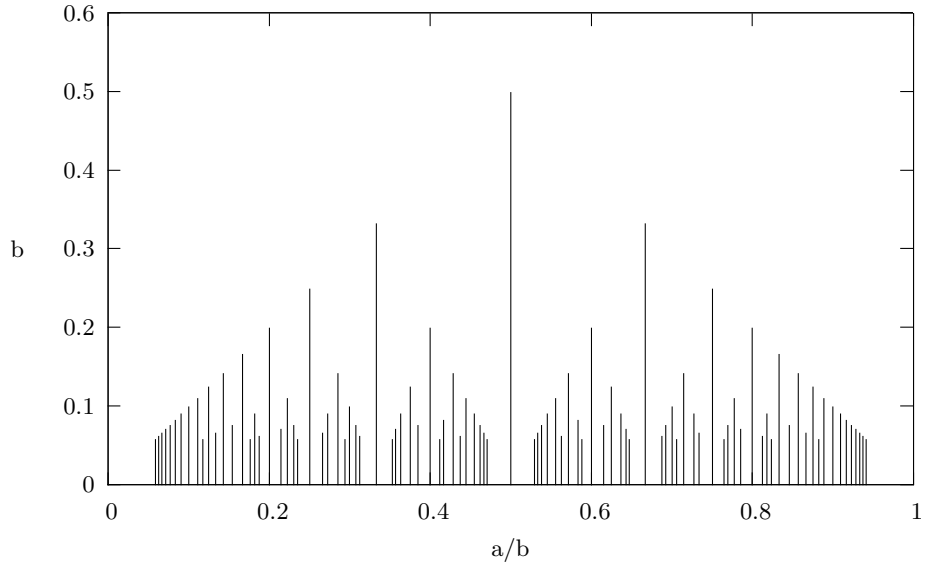


Fig. 7. Correlation of $a/b \in F_{17}$ and $1/b$ in the interval $(0, 1)$

the first music application for composition where rhythms and musical forms will be generated on the basis of the principles outlined in this paper. The Farey Sequence F_n is per se highly symmetrical and unfolds harmonic subdivisions of unity via a recursive calculation of mediant fractions[9]. Every music that depends on an underlying beat or pulsation can be represented by using F_n to denote the normalised occurrence of musical events, e.g. note onsets.

Off-Beat rhythms are extremely useful for generative purposes. Two or more of these rhythms stacked in layers can always generate new combinations by using different accentuation patterns and different dynamic processes. Again, the Farey-Sequence proves to be a very useful structure to realise this concept. The details are beyond what is necessary in this paper, but in **?**) it is shown that many styles, Bebop, Funk and 20th century Avantgarde, are modelled by this mechanism. Speech rhythm, as used in various musical styles are also within the scope.

Finally, Sima Arom's study (Arom 1991) on African Polyrhythms has been very influential on contemporary western composers because of his successful recording and transcription processes that form the basis of his further analysis. We are seeking to translate some of these principles into features for ANTON for creative purposes but also in order to proof the general use of our concept. But this is for the future (section 8).

The main message is that the Farey sequence contains all that is necessary for a very wide range of rhythmic patterns. With an implementation of them within the ASP framework we have all the infrastructure we need.

[9] see http://mathworld.wolfram.com/FareySequence.html

```
%% Each Farey tree has a given depth
depth(F,MD + BD + DD) :- measureDepth(MD), beatDepth(F,BD), durationDepth(F,DD).
level(F,1..DE) :- depth(F,DE).
%% Each Farey tree is divided into three layers (top to bottom)
%% Measure, beats and note duration
%% (bars, time signature and note value)
measureLevel(F,FL) :- depth(F,DE), durationDepth(F,DD), beatDepth(F,BD),
                      level(F,FL), FL <= DE - (DD + BD).
measureLeafLevel(F,DE - (DD + BD)) :- depth(F,DE), durationDepth(F,DD), beatDepth(F,BD).
%% Beat strength is created at the first level of the beat layer ...
nodeBeatStrength(F,MLL+1,ND2,1) :- measureLeafLevel(F,MLL), node(F,MLL,ND1),
                      descendant(F,0,MLL,ND1,MLL+1,ND2).
nodeBeatStrength(F,MLL+1,ND2,0) :- measureLeafLevel(F,MLL), node(F,MLL,ND1),
                      descendant(F,D,MLL,ND1,MLL+1,ND2), D != 0.

%% Map from nodes to time positions
%% Mapping increments each time a node is present
nodeStep(F,0,1).
nodeStep(F,ND,T) :- not present(F,DLL,ND), nodeStep(F,ND-1,T),
                    node(F,DLL,ND), durationLeafLevel(F,DLL), ND > 0.
nodeStep(F,ND,T+1) :- present(F,DLL,ND), nodeStep(F,ND-1,T),
                      node(F,DLL,ND), durationLeafLevel(F,DLL), ND > 0.
%% From this we derive a unique mapping from node to time step
timeToNode(P,1,0).
timeToNode(P,T,ND) :- present(F,DLL,ND), nodeStep(F,ND-1,T-1),
node(F,DLL,ND), durationLeafLevel(F,DLL), ND > 0,
partToFareyTree(P,F).
%% Beat strength is created at the first level of the beat layer ...
nodeBeatStrength(F,MLL+1,ND2,1) :- measureLeafLevel(F,MLL), node(F,MLL,ND1),
descendant(F,0,MLL,ND1,MLL+1,ND2).
nodeBeatStrength(F,MLL+1,ND2,0) :- measureLeafLevel(F,MLL), node(F,MLL,ND1),
descendant(F,D,MLL,ND1,MLL+1,ND2), D != 0.

%% Lowest and highest notes must also be slower
playsHeighestNote(P,T) :- choosenNote(P,T,N), lowestNote(P,N).
playsLowestNote(P,T) :- choosenNote(P,T,N), lowestNote(P,N).
:- playsHeighestNote(P,T), timeStepDuration(P,T,DS), DS > 1.
:- playsLowestNote(P,T), timeStepDuration(P,T,DS), DS > 1.
```

Fig. 8. A small rhythm code fragment

The question now arises how we can combine these generative rules for rhythms with the rules for melody, counterpoint and harmony that have been already implemented in ANTON. Of central importance for musical experiences in our view is the constant inter-change of *impact* and *resolution* that influences the behaviour of musical parameters on micro- and macro-structural levels. One can compare *impact* with gathering musical energy (Thakar 1990) and *resolution* with the release of previously built-up energy. We were very careful to make sure that the melodic lines generated are following this principle. With encoding of rhythms we have now the possibility to precisely control the timing aspect of when to turn the musical movement from impact to resolution.

5.2 Encoding

The encoding of rhythm is still in an experimental phase and will improve over time. As mentioned earlier, the encoding of rhythm is based on the concept of Farey trees. The encoding can be found in rhythm.lp. In ANTON 1.5 the system only creates an Farey tree and imposes it on the system. The newest version, which is not yet released, adds rules to deal with beats, their strength, duration and the interplay between parts.

Figure 8 contains a small code fragment of the current rhythm section of the

system. The fragments shows a portion of the structure of the Farey tree and its three layers. Each of these have separate rules that govern them. The fragment also includes to mapping from notes to time instances. We have included one of the rules on beat strength and one on the relation between tone and speed.

While not shown in this code fragment, the rhythm code is, like the other sections, written in such a way that diagnosis and debugging are possible.

In order to include rhythm in a composition, it suffices the include options –rhythm to the programBuilder.pl script. The command

```
./programBuilder.pl --task=compose --mode=lydian --time=12 --style=duet
--rhythm > rhythm-comp
```

composes a duet in Lydian with 12 notes per part.

In the newest version one can also include measures.

The call

```
./programBuilder.pl --task=compose --mode=minor --time=20 --style=duet
--measure=3 --rhythm > rhythm-comp
```

composes a quartet in minor with 20 notes per part split over 3 measures.

The program can then be parsed using parse.lp to generate Csound, Lilypond, Graphviz, human readable output. Figure 9 shows the Farey tree and the human score for Lydian duet.

6 Evaluation

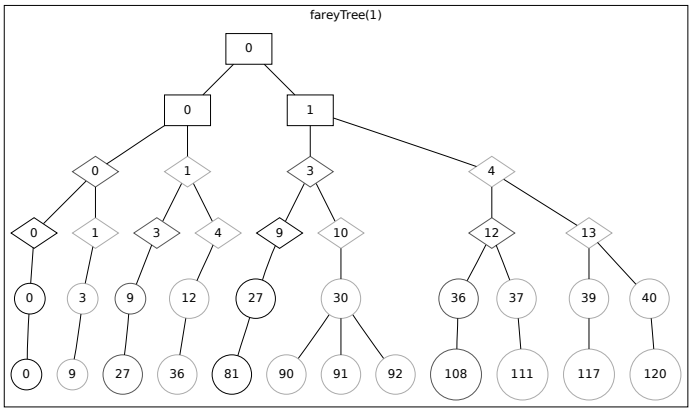
In this section we evaluate the performance of ANTON both from a system's perspective and a musical perspective. While we mainly focus on the released version 1.5 for stability and reproducibility, we will also touch on the performance of the newer version to identify outstanding issues.

6.0.1 Run-time Results

To evaluate the practicality of using answer set programming in a composition system we timed ANTON 1.5 without the rhythm version while composing a suite of score with increasing difficulty.

Tables 3-6 contains the timings for a number of answer set solvers (CLASP (Gebser et al. 2007b), CMODELS (Lierler and Maratea 2004), SMODELS (Syrjänen and Niemelä 2001), SMODELS-IE (Brain et al. 2007), SMODELSCC (Ward and Schlipf 2004), and SUP (Lierler 2008)) in composing solos, duets, trios and quartets of a given length.

Benchmarks were run using a 2.4Ghz AMD Athlon X2 4600+ processor, running a 64 bit version of OpenSuSE 11.1. All solvers were built in 32 bit mode. Each test was limited to 10 minutes of CPU time and 2Gb of RAM. Programs were ground using GRINGO 2.0.3 and grounding times were excluded from the reported times, but typically were a few seconds at most. All solvers were run using default options, except CMODELS which was set to use the MINISAT 2.0 back end as opposed to the




```
| 65 67 65 77 76 69 71 76 72 74 76 77
|  F  G  F  F' E' A  B  E' C' D' E' F'
|   +2 -2 +12 -1 -7 +2 +5 -4 +2 +2 +1
| (((X X) (X X)) ((X (X X X)) ((X X) (X X))))

| 65 60 62 62 60 65 67 67 69 67 67 65
|  F  C  D  D  C  F  G  G  A  G  G  F
|  -5 +2 "" -2 +5 +2 "" +2 -2 "" -2
| (((X X) (X X)) ((X (X X X)) ((X X) (X X))))
```

Fig. 9. The Farey tree and human readable format for the tunes composition.

default (ZCHAFF). The programs used are available from `http://www.cs.bath.ac.uk/~mjb/anton`.

The results show a significant increase in performance from the ANTON1.0 version reported in (Boenn et al. 2008). In (Boenn et al. 2008), we were only able to compose duets up to length 16, which took 29.63 seconds using the fastest solver CLASP. The current system, ANTON1.5, only takes 1.01 seconds for the same composition using the same solver. Furthermore, we can now compose trios and quartets within a reasonable time frame.

While improvements in the underlying solvers definitely contributed to steep increase in performance, they are not the main contributor. We obtained most of our increases by revisiting our encoding and finding more compact encodings. We have compacted the rule set of minor keys which gives some reduction is space and run time. We reformatted some of the harmony rules and relaxed them so they only apply to neighbouring parts rather than all parts. The removal of redundant constraints compacts the program by a surprising amount. The rewriting of the repeated notes section produced a massive increase in grounding while the improved encoding of highest and lowest note saved us about 150,000 grounded rules on 16 note duet and about 30% in run time. Ranges over two octaves is now only noted

	Solvers					
Length	Clasp 1.2.1	Cmodels 3.79	Smodels 2.33	Smodels-IE 1.0.0	Smodels_cc 1.08	Sup 0.4
4	0.02	0.09	0.04	0.02	0.10	0.04
8	0.18	0.38	1.27	0.52	4.55	0.16
12	0.48	1.10	8.99	2.87	27.45	0.64
16	1.05	2.06	36.03	10.19	86.56	1.01
20	2.68	3.11	32.52	10.02	93.61	2.06
24	2.42	4.22	193.40	58.06	Time out	2.11
28	3.84	5.80	239.49	80.56	Time out	4.02
32	3.90	7.11	305.05	102.91	Time out	4.66

Table 3. *Time taken (in seconds) for a number of solvers generating a solo piece.*

	Solvers					
Length	Clasp 1.2.1	Cmodels 3.79	Smodels 2.33	Smodels-IE 1.0.0	Smodels_cc 1.08	Sup 0.4
4	0.14	0.28	0.23	0.10	0.64	0.14
8	0.43	0.98	10.60	4.92	77.07	0.57
12	2.28	2.54	Time out	Time out	Time out	2.18
16	1.91	3.92	Time out	Time out	Time out	3.33
20	3.12	6.58	Time out	Time out	Time out	7.86
24	8.60	8.71	Time out	Time out	Time out	13.55
28	14.94	19.29	Time out	Time out	Time out	31.05
32	13.56	26.90	Time out	Time out	Time out	31.63

Table 4. *Time taken (in seconds) for a number of solvers generating a duet piece.*

at the end of the program, rather than at the point at which it is triggered. While this is slightly less informative, it offers a more than significant speed-up.

These results show that the system, using the the more powerful solvers, is not only fast enough to be used as a component in an interactive composition tool but, when restricting to shorter sequences, could be used for real-time generation of music.

	Solvers					
Length	Clasp 1.2.1	Cmodels 3.79	Smodels 2.33	Smodels-IE 1.0.0	Smodels_cc 1.08	Sup 0.4
4	0.26	0.30	0.27	0.04	0.30	0.23
8	0.78	1.52	3.42	1.34	10.39	0.96
12	2.17	3.20	22.44	8.52	77.80	2.88
16	2.81	5.35	104.68	39.20	Time out	4.55
20	6.95	7.63	Time out	Time out	Time out	9.01
24	11.90	10.92	Time out	Time out	Time out	9.75
28	41.28	12.32	Time out	Time out	Time out	10.68
32	52.98	19.33	Time out	Time out	Time out	47.94

Table 5. *Time taken (in seconds) for a number of solvers generating a trio piece.*

	Solvers					
Length	Clasp 1.2.1	Cmodels 3.79	Smodels 2.33	Smodels-IE 1.0.0	Smodels_cc 1.08	Sup 0.4
4	0.43	0.55	0.49	0.08	0.63	0.55
8	1.14	2.55	7.98	3.28	32.20	2.70
12	4.24	5.38	35.11	13.07	128.75	5.46
16	9.59	8.64	336.36	126.21	Time out	16.86
20	69.37	11.87	Time out	Time out	Time out	17.44
24	194.73	20.44	Time out	Time out	Time out	30.99
28	246.10	19.32	Time out	Time out	Time out	79.13
32	Time out	46.61	Time out	Time out	Time out	113.30

Table 6. *Time taken (in seconds) for a number of solvers generating a quartet piece.*

It is also interesting to note that the only solvers able to generate longer sequences using two or more parts all implement clause learning strategies, suggesting that the problem is particularly susceptible to this kind of technique.

We have not included run-time results for the rhythm section as this part is still too much under development and the results would not be representative.

6.0.2 Music Quality

The interested reader can find examples on the web: `http://dream.cs.bath.ac.uk/Anton`

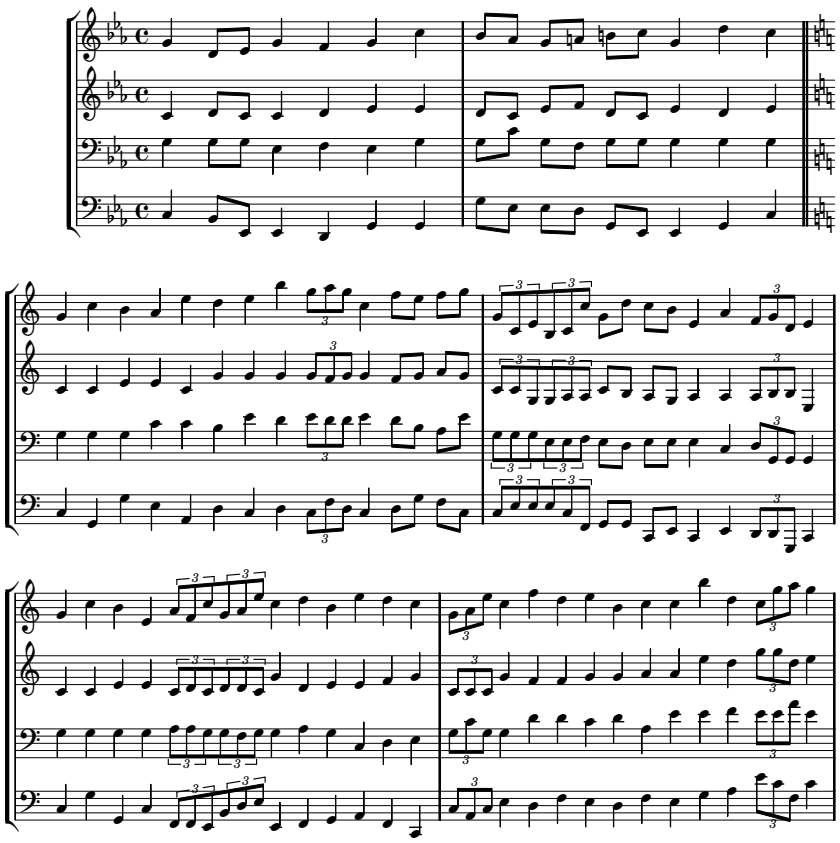


Fig. 10. Fragments by ANTON

The other way to evaluate the system is to judge the music it produces. This is less certain process, involving personal values. However we feel that the music is acceptable, at least of the quality of a student of composition, and at times capable of moments of excitement. Pieces by ANTON1.0 have been played to a number of musicians, who apart from the rhythmic deficiency we are addressing have agreed that it is valid music. The introduction of rhythm is more recent, and consequently it has not been subjected to so much scrutiny. There are still some refinements that could improve the output, in particular an unfortunate tendency to end on shorter

notes, but many of the short pieces are clearly valid, and musical. In figure 10 we present a short quartet sequence in the minor key, followed by four major key pieces that ANTON composed especially for this paper; the audio and score can be found in the same location as the other works.

7 The Use of ASP in Anton

7.1 Why ASP?

While music appreciation is matter of personal taste, musicologists use sets of rules which determine to which style a musical composition belongs or whether a piece breaks or expands the common practice of a certain composer or era. These sets of rules also govern the composition. So an intuitive and obvious way for automatic composition is to encode these rules and use a rule based algorithm to produce valid music compositions. This natural and simple way of encoding things is show in terms of speed of development, roughly 2 man-months, sophistication of the results, the amount of code (about 200 lines of code) and flexibility; we can not only easily encode different styles but the same application not only for automated composition but also diagnosis and human assisted composition. Furthermore, we automatically gain from any improvements in the underlying solver.

7.2 ASP a the Knowledge Representation Language

For ANTON we used ASP for synthesis rather than problem solving. Normally refer to ASP as a problem solving paradigm which it is. In this case we are doing something subtly, but importantly different; we are using a solver to generate representative objects given a specification. This is more knowledge presentation, since the language is not only used to describe the objects but also to come up with a computational description. We believe that this opens a new range of possibilities for ASP as there are lots of applications for we need parametrisable, consistent if not spectacular contents. Games, virtual worlds and puzzle magazines are just a few examples.

7.3 ASP Methodology

In constructing ANTON a number of advantages of using answer set programming have become clear, as have a number of limitations.

Firstly, ASP programs are very fast to write and very compact. As well as the obvious benefits, this means it is possible to develop the system at the same time as undertaking knowledge capture and to prototype features in the light of the advice of domain experts. Part of the reason why it is so fast to use is that rules are fully declarative. Programming thus focuses on expressing the concepts that are being modelled rather than having to worry about which order to put things in — such as which rules should come first, which concepts have higher priority, which choices should be made first. This also makes incremental development easy as new

constraints can be added one at a time, without having to consider how they affect the search strategy.

Being able to add rules incrementally during development turns out to be extremely useful from a software engineering view point. During the development of ANTON, we experimented with a number of different development methodologies. As argued in (Cliffe et al. 2008), "visualisation" of answer sets is very productive way bridging the gap between domain and the program. For the musical application domain, the most effective approach was found to be first writing a script that translates answer sets to human readable score or output for a synthesiser. Next the choice rules were added to the program to create all possible pieces, valid or not. Finally the constraints were incrementally added to restrict the output to only valid sequences. By building up a library of valid pieces it was possible to perform regression testing at each step and thus isolate bugs as soon as they were introduced.

Using answer set programming was not without issue. One persistent problem was the lack of mature development support tools, particularly debugging tools. SPOCK (Brain et al. 2007) was used but as its focus is on computing the reasons behind the error, rather than the interface issues of explaining these reasons to the user, it was normally quicker to find bugs by looking at the last changes made and which regression tests failed. Generally, the bugs that where encountered where due to subtle mismatches between the intended meaning of a rule and the declarative reading of the rule used. For example the predicate `stepUp(P,T)` is used to represent the proposition "At time T, part P steps up to give the note at time T+1", however, it could easily be misinterpreted as "At time T-1, part P steps up to give the note at time T". Which of these is used is not important, as long as the same declarative reading is used for all rules. With the first "meaning" selected for ANTON, the rule:

```
chosenNote(P,T,N+S) :-  chosenNote(P,T-1,N), stepUp(P,T),
                        stepBy(P,T,S).
```

would not encode the intended progression of notes. One possible way of supporting a programmer in avoiding these subtle errors would be to develop a system that translated rules into natural language, given the declarative reading of the propositions involved. It should then be relatively straightforward to check that the rule encoded what was intended.

8 Conclusions and Future Work

We have presented an algorithmic composition system that uses ASP to implement rules captured from classical texts on species of one counterpoint. The early system of Boenn et al. (2008) has been developed in two significant ways: greatly improved performance and the introduction of a coherent rhythm schema.

The development of ANTON is far from complete.

The encoding of rhythm is not finished. Although we now have a basic encoding of rhythm it still requires a lot of fine tuning. The use of Farey trees is the core

of this. In Section 5 we have already indicated an area of study where the use of Farey trees can be further developed.

We have concentrated on species one counterpoint from an early time, but many of the rules apply to other styles. We have made a few experiments with rules for Bach chorales and for hymn tunes. We need to partition the current rule-sets into building blocks to facilitate reuse.

The current system can write short melodies effectively and efficiently. Development work is still needed to take this to entire pieces; we can start from these melodic fragments but a longer piece needs a variety of different harmonisations for the same melody, and related melodies with the same harmonic structure and a number of similar techniques. We have not solved the difficult global structure problem but it does create a starting point on which we can build a system that is hierarchical over time scales; we have a mechanism for building syntactically correct sentences, but these need to be built into paragraph and chapters, as it were. It is not clear if this will be achieved within the current ASP system, or by a procedural layer, or some other scheme.

To make the system more user-friendly, there is a need for a user interface, probably graphical, to select from the options and styles. We have avoided this so far, as the Musician-Machine Interface is a specialist area, but there are plans for such an interface to be designed in the next phase.

In real life pieces some of the rules are sometimes broken. This could be simulated by one of a number of extensions to answer set semantics (preferences, consistency restoring rules, defensible rules, etc.). However how to systematise the knowledge of when it is acceptable to break the rules and in which contexts it is 'better' to break them is an open problem.

As mentioned earlier, we used ASP as a computational description language rather than just a knowledge representation one. There is a tempting possibility to apply the same methodology and approach to other areas of content, such as maybe game map generation. Initial experiments show promise.

References

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *CACM 26*, 198–3.

Anders, T. 2007. Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System. Ph.D. thesis, Queen's University, Belfast, Department of Music.

Arom, S. 1991. *African polyphony and polyrhythm.* Cambridge University Press, Cambridge.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge Press.

Baral, C. and Gelfond, M. 2000. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence.* Kluwer Academic Publishers, 257–279.

Bel, B. 1998. Migrating Musical Concepts: An Overview of the Bol Processor. *Computer Music Journal 22,* 2, 56–64.

Boenn, G. 2007. Composing Rhythms Based Upon Farey Sequences. In *Digital Music Research Network Conference.*

Boenn, G. 2008. The importance of husserl's phenomenology of internal time-consciousness for music analysis and composition. In *Proceedings of the ICMC 2008*. Belfast.

Boenn, G., Brain, M., De Vos, M., and ffitch, J. 2008. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. In *International Conference on Logic Programming, ICLP08*. Lecture Notes in Computer Science, vol. 4386. Springer Berlin / Heidelberg, 160–174. in print.

Boulanger, R., Ed. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press.

Brain, M., Crick, T., De Vos, M., and Fitch, J. 2006. TOAST: Applying Answer Set Programming to Superoptimisation. In *International Conference on Logic Programming*. LNCS. Springer.

Brain, M., De Vos, M., and Satoh, K. 2007. Smodels-ie : Improving the Cache Utilisation of Smodels. In *Proceedings of the 4th Workshop on Answer Set Programming*, S. Costantini and R. Watson, Eds. 309–314.

Brain, M., De Vos, M., and Satoh, K. 2007. smodels-ie: Improving the cache utilisation of smodels. In *Proceedings of the 4th Workshop on Answer Set Programming: Advances in Theory and Implementation*. Porto, Portugal, 309–313.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. 2007. "That is illogical captain!" – The debugging support tool spock for answer-set programs: System description. In *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, M. De Vos and T. Schaub, Eds. 71–85.

Brothwell, A. and ffitch, J. 2008. An Automatic Blues Band. In *6th International Linux Audio Conference*, F. Barknecht and M. Rumori, Eds. Tribun EU, Gorkeho 41, Bruno 602 00, Kunsthochscule für Medien Köln, 12–17.

Buccafurri, F. and Caminiti, G. 2005. A social semantics for multi-agent systems. In *LPNMR*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer, 317–329.

Buccafurri, F. and Gottlob, G. 2002. Multiagent compromises, joint fixpoints, and stable models. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, A. C. Kakas and F. Sadri, Eds. Lecture Notes in Computer Science, vol. 2407. Springer, 561–585.

Chuang, J. 1995. Mozart's Musikalisches Würfelspiel. `http://sunsite.univie.ac.at/Mozart/dice/`.

Cliffe, O., De Vos, M., Brain, M., and Padget, J. 2008. Aspviz: Declarative visualisation and animation using answer set programming. In *Logic Programming*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 724–728.

Cliffe, O., De Vos, M., and Padget, J. 2006. Specifying and Analysing Agent-based Social Institutions using Answer Set Programming. In *Selected revised papers from the workshops on Agent, Norms and Institutions for Regulated Multi-Agent Systems (ANIREM) and Organizations and Organization Oriented Programming (OOOP) at AAMAS'05*, O. Boissier, J. Padget, V. Dignum, G. Lindemann, E. Matson, S. Ossowski, J. Sichman, and J. Vazquez-Salceda, Eds. LNCS, vol. 3913. Springer Verlag, 99–113.

Cope, D. 2006. A Musical Learning Algorithm. *Computer Music Journal 28,* 3 (Fall), 12–27.

de la Motte, D. 1981. *Kontrapunkt*. dtv/Bärenreiter, München.

De Vos, M. and Vermeir, D. 1999. Choice Logic Programs and Nash Equilibria in Strategic Games. In *Computer Science Logic (CSL'99)*, J. Flum and M. Rodríguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 1683. Springer Verslag, Madrid, Spain, 266–276.

De Vos, M. and Vermeir, D. 2004. Extending Answer Sets for Logic Programming Agents. *Annals of Mathematics and Artifical Intelligence 42,* 1–3 (Sept.), 103–139. Special Issue on Computational Logic in Multi-Agent Systems.

Ebcioğlu, K. 1986. An Expert System for Harmonization of Chorales in the Style of J.S. Bach. Ph.D. thesis, State University of New York, Buffalo, Department of Computer Science.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. 1999. The diagnosis frontend of the dlv system. *AI Communications 12,* 1-2, 99–111.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. 2002. The DLV^K Planning System. In *European Conference, JELIA 2002*, S. Flesca, S. Greco, N. Leone, and G. Ianni, Eds. LNAI, vol. 2424. Springer Verlag, Cosenza, Italy, 541–544.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. 1998. The KR system `dlv`: Progress report, comparisons and benchmarks. In *KR'98: Principles of Knowledge Representation and Reasoning*, A. G. Cohn, L. Schubert, and S. C. Shapiro, Eds. Morgan Kaufmann, San Francisco, California, 406–417.

Erdem, E., Lifschitz, V., Nakhleh, L., and Ringe, D. 2003. Reconstructing the Evolutionary History of Indo-European Languages Using Answer Set Programming. In *PADL*, V. Dahl and P. Wadler, Eds. LNCS, vol. 2562. Springer, 160–176.

Erdem, E., Lifschitz, V., and Ringe, D. 2006. Temporal phylogenetic networks and logic programming. *TPLP 6,* 5, 539–558.

Fux, J. 1965, orig 1725. *The Study of Counterpoint from Johann Joseph Fux's Gradus ad Parnassum.* W.W. Norton.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007a. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/The MIT Press, 386–392. Available at http://www.ijcai.org/papers07/contents.php.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007b. Conflict-Driven Answer Set Solving. In *Proceeding of IJCAI07*. 386–392.

Gebser, M., Schaub, T., and Thiele, S. 2007. Gringo: A new grounder for answer set programming. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer-Verlag, 266–271.

Gelfond, M. and Lifschitz, V. 1988a. The stable model semantics for logic programming. See Kowalski and Bowen (1988), 1070–1080.

Gelfond, M. and Lifschitz, V. 1988b. The stable model semantics for logic programming. See Kowalski and Bowen (1988), 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3-4, 365–386.

Giunchiglia, E., Lierler, Y., and Maratea, M. 2004. SAT-Based Answer Set Programming. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-04)*. 61–66.

Goranko, V., Montanari, A., and Sciavicco, G. 2003. A road map on interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics 14*, 9–54.

Graham, R. L., Knuth, D. E., and Patashnik, O. 1990. *Concrete Mathematics.* Addison-Wesley.

Grell, S., Schaub, T., and Selbig, J. 2006. Modelling biological networks by action languages via answer set programming. In *Proceedings of the International Conference on Logic Programming (ICLP'06)*, S. Etalle and M. Truszczyński, Eds. LNCS, vol. 4079. Springer-Verlag, 285–299.

Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S., and Tichy, R. 2005. Platypus: A Platform for Distributed Answer Set Solving. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*. 227–239.

Hajdu, G. 1993. Low energy and equal spacing. the multifactorial evolution of tuning systems. *Interface 22*, 319–333.

Hardy, G. and Wright, E. 1938. *An Introduction to the Theory of Numbers*, 4th ed. Oxford University Press.

Konczak, K. 2006. Voting Theory in Answer Set Programming. In *Proceedings of the Twentieth Workshop on Logic Programming (WLP'06)*, M. Fink, H. Tompits, and S. Woltran, Eds. Number INFSYS RR-1843-06-02 in Technical Report Series. Technische Universität Wien, 45–53.

Kowalski, R. A. and Bowen, K. A., Eds. 1988. *Logic Programming, Proceedings of the Fifth International Conference and Symposium*. The MIT Press, Seattle, Washington.

Leach, J. L. 1999. Algorithmic Composition and Musical Form. Ph.D. thesis, University of Bath, School of Mathematical Sciences.

Lierler, Y. 2008. Abstract answer set solvers. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*. Springer-Verlag, Berlin, Heidelberg, 377–391.

Lierler, Y. and Maratea, M. 2004. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*. LNCS, vol. 2923. Springer, 346–350.

Lifschitz, V. 2002. Answer set programming and plan generation. *J. of Artificial Intelligence 138,* 1-2, 39–54.

Mileo, A. and Schaub, T. 2006. Extending Ordered Disjunctions for Policy Enforcement: Preliminary report. In *Proceedings of the International Workshop on Preferences in Logic Programming Systems (PREFS'06)*, E. Pontelli and T. Son, Eds. 45–59.

Niemelä, I. and Simons, P. 1997. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, J. Dix, U. Furbach, and A. Nerode, Eds. LNAI, vol. 1265. Springer, Berlin, 420–429.

Niemelä, I., Simons, P., and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *LPNMR*, M. Gelfond, N. Leone, and G. Pfeifer, Eds. Lecture Notes in Computer Science, vol. 1730. Springer, 317–331.

Padovani, L. and Provetti, A. 2004. Qsmodels: Asp planning in interactive gaming environment. In *JELIA*, J. J. Alferes and J. A. Leite, Eds. Lecture Notes in Computer Science, vol. 3229. Springer, 689–692.

Partch, H. 1979. *Genesis of a Music*. Da Capo Press, New York.

Rohrmeier, M. 2006. Towards modelling harmonic movement in music: Analysing properties and dynamic aspects of pc set sequences in Bach's chorales. Tech. Rep. DCRR-004, Darwin College, University of Cambridge.

Soininen, T. and Niemelä, I. 1999. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*. LNCS. Springer, San Antonio, Texas.

Syrjänen, T. and Niemelä, I. 2001. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*.

Thakar, M. 1990. *Counterpoint*. New Haven.

Ward, J. and Schlipf, S. 2004. Answer Set Programming with Clause Learning. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*. LNCS, vol. 2923. Springer.

Xenakis, I. 1992. *Formalized Music.* Bloomington Press, Stuyvesant, NY, USA.